6.0 SOFTWARE QUALITY ASSESSMENT

The purpose of the analysis described in this report is to support the verification of the BRAWLER air combat simulation by analyzing the structure of its source code from a software engineering perspective. The evaluation method was a static examination of source code of selected routines. No attempt was made in this analysis to run CASE tools on the software, to perform runtime analysis, or to analyze or evaluate BRAWLER output.

6.1 MODEL DESCRIPTION

As with any software system with an active user base, BRAWLER has been released numerous times in many versions over a long period of time. The information in this section describes the latest release version that was the subject of this analysis. Table 6-1 lists the pertinent release information of the version analyzed.

TABLE 6-1. BRAWLER Release Specifics.

NAME:	BRAWLER
ACRONYM:	none
VERSION:	6.2
DATE OF RELEASE:	31 OCT 1995

6.1.1 Functional Description

BRAWLER is a Monte Carlo, event-driven simulation of flight-versus-flight air combat. Inherent within the simulation is the explicit modeling of human decision processes as well as the modeling of:

a. Aircraft, including aerodynamics and signatures.

b. Weapons systems, including missile models, guns, and fire control characteristics.

c. Avionics, including radars, infrared search and track (IRST), missile launch warning devices, radar warning receiver (RWR), identification friend-or-foe (IFF), and non-cooperative identification (NCID) techniques.

d. Countermeasures, including expendables and electronic countermeasures versus radars, missiles, and communications.

In addition to aircraft systems, BRAWLER provides simple surface-to-air missile (SAM) and air-to-air (AAM) models.

6.1.2 Source Code Statistics

The BRAWLER program comprises 16 directories containing the source code, common blocks, utilities and data files for the model. Five of the directories (convert, secret, source, uroutines, and utils) contain source code for executable routines. There are five large routines that contain more than 28,000 bytes and 62 small routines that are less than 100 bytes in size. Three routines, fpisig.c, iand.f, and trk_vrs_x.f, are 0 bytes in size - they

are module titles only, containing no coding or comments. The largest routine, *convert85.f*, is 146,748 bytes in size, contains 8 other subroutines, and implements multiple missile engine capability. The second largest routine, *disclientlib.c*, is 70,948 bytes in size and is a utility program written in C that opens and closes communication sockets with distributed interactive simulations (DIS) and reads or writes protocol data units (PDU) to the socket. The routine average size is 4371 bytes which is about three pages in length, with approximately one page being header and comments. Twelve of the small routines were selected for examination and were found to be null or dummy routines that contained no application coding. Most of the dummy routines exhibited the poor programming practice of not including a "return" statement or a header explaining why they are dummies. Three routines, *dog_sync.c*, with a size of 490 bytes, *calm.f* with 588 bytes, and *rrthrt.f* with 643 bytes were also dummies. The larger sizes of these null routines was due to headers that were more extensive than those in the small dummy routines. There are 206 routines that are less than 600 bytes in size and it is possible that many of them are dummy routines. Appendix A contains a listing of source code file names and sizes for the main programs, subroutines, and functions that comprise BRAWLER V6.2. The routines that were examined have an '*' after their source code file name. The dummy routines are identified with a '**' and were not utilized in the numerical evaluations. Appendix B lists the specific routines that implement functional elements within functional area templates. All of the functional element routines in the list were examined. However, all the subsequent routines that are called by the functional area routines were not examined. Appendix C contains the software analysis worksheets for the routines that were evaluated and Appendix D has listings of two representative routines and one function.

6.2 CODE STRUCTURE ANALYSIS AND CRITERIA

6.2.1 Organization

BRAWLER is a large model, with more than 200,000 lines of executable source code, consisting of nine main program modules and over 3,679 subroutines, functions, include files and other forms of subordinate modules. The exact number of routines was not determined because some routines contain embedded routines and functions (e.g., *cvrt_ir_exec.f* contains 10 subroutines and ned_to_dis.c contains 6 routines and functions) and all subroutine files were not examined. Thus, it is possible that several other routines contain embedded routines. The overall structure of BRAWLER is very modular with most called subroutines contained in individual files. Of the routines identifiable as source code, 81 are in C and the rest are in FORTRAN. The flight tactics decision algorithms are probably unfamiliar to most people and a programmer/analyst should have good familiarity with them and the operating system and its utilities before attempting to make any program modifications. The version of BRAWLER examined for this study was UNCLASSIFIED.

As a result of the size of BRAWLER, and time and resource limitations, the analysis focused on a sample set of 123 subroutines. The routines were not selected on a totally random basis. The five largest routines (more than 28,000 bytes in size) were selected first for examination. Then a few smaller routines were selected. The results from this sample were then augmented by selecting additional modules that perform various flight decision logic processes. Finally, the routines comprising the functional areas were selected. Excluding the dummy routines, 111 routines were numerically evaluated. Even though this sample represents a small sample of the total code, it should provide a reasonable indication

of the construction of the simulation as a whole since the developer's software quality philosophy provides general guidance for development of the entire simulation.

6.2.2 Evaluation Criteria

The subjective nature of quality evaluations combined with personal programming styles and preferences can result in different assessments of the same code from different analysts. The balance of this section describes the four measures of effectiveness (MOEs) chosen for this SQA and attempts to describe an appropriate set of contributing factors (criteria) for each MOE. The descriptions of evaluation criteria should allow the reader to reference his preferences to those of the analyst performing the SQA.

6.2.3 MOE #1 Use of Standards

6.2.3.1 Readability

Programmer style varies, but there should be certain consistencies from routine to routine and they should be organized in a similar order. There should be a header, version number, purpose, variable declaration, include statements, all with appropriate comments in a format that is pleasing to the eye. Code should be neat and clear with appropriate comments and line spacing as the programmer sees fit to make the code easy to read.

6.2.3.2 Modifiability

A well designed, modular structure that adheres to good software development practices, with meaningful variable names and descriptive comments will allow a model to be easily modified, updated or expanded to include additional features or capabilities. A high score in modifiability is dependent on high scores in other criteria (e.g., readability, variable declarations, naming conventions).

6.2.3.3 ANSI standards

Each module analyzed was compared to the American National Standard FORTRAN-77 X3.9-1978 standard for compliance. This standard specifies the form and establishes the interpretation of programs expressed in the FORTRAN language. The purpose of this standard is to promote portability of FORTRAN programs for use on a variety of data processing systems. The requirements, prohibitions, and options specified in this standard, generally refer to permissible forms and relationships for standard conforming programs.

6.2.4 MOE #2 Programming Conventions

6.2.4.1 Use of Comments and Headers

Comments should consist of large descriptive headers containing the version number of the source code complete with authors name and dates that indicate when the code was written with a revision history. This is not required but recommended. The header should also contain a technical description defining the routines purpose and functionality. Variable names, local and global should be listed, one to a line with clear definitions. Each subroutine that is called should also be listed with a short description of its intent. Embedded comment should be plentiful, clear and adequately provide a good understanding of what is intended. Classification markings, top and bottom will also be considered.

6.2.4.2 Use of Formatted Statements

The source code should be formatted to aid readability and modifiability. Variable declarations, COMMON, TYPE and DATA statements should be located at the beginning of the module, FORMAT and other non-executable statements should be at the end of the module. Readability is improved if embedded comments are written in lower case with blank lines appropriately placed to highlight certain areas. DO loops and IF blocks need to be indented to improve readability and understandability.

6.2.4.3 Logical I/O Devices

All file I/O units should be initialized as variables with no reference to specific units by number. The variable names should be meaningful and properly commented to facilitate understanding.

6.2.4.4 Variable Declarations

The term variable is used to denote stored data items represented by symbolic names associated with a storage location. Variables are classified by data type. Type declaration statements explicitly define the data type of specified symbolic names. There are two forms of type declaration statements: numeric type declaration (byte, logical, integer, real, double precision, complex, and double complex) and character. Defaults will not be allowed, and declared variables without comments will be rated poorly. The modules should all contain the statement IMPLICIT NONE. The use of IMPLICIT NONE forces the programmer to define each and every one of the variables in use before it is used. As a result, no default variable types are accepted. This allows the code to be more easily understood and modified.

6.2.4.5 Variable Initialization

Initializations may be performed in many ways in FORTRAN. Assignment statements (e.g., VALUE=NUMBER), DATA statements, DIMENSION statements, and EQUIVALENCE statements may all be used; however, the use of the latter is generally considered poor practice. The expression NUMBER should be evaluated and verified that it conforms to the range requirements of VALUE. Comments, as always, will contribute to a higher rating.

6.2.4.6 Variable Naming Conventions

A well defined naming convention for variables should be established before coding begins. Consistent, meaningful variable names with comments aid in the readability and modifiability of a module.

6.2.4.7 Algorithm Clarity

Algorithms should be developed in a modular fashions. Long complicated equations should be broken up into smaller, readable well commented sections. Each element in the calculation should be executed on its own line if possible. No attempt was made to verify the correctness of the algorithms examined.

6.2.5 MOE #3 Computational Efficiency

6.2.5.1 Mixed Mode Calculations

Combining integers and real numbers in a calculation, although not prohibited, can result in errors due to the different internal representation of the two variable types if used incorrectly. In general this is considered a poor programming practice and should not be used without sufficient comments that describe why the author feels it is necessary in the computation.

6.2.5.2 Use of Library Functions

FORTRAN library function names are called intrinsic function names. Intrinsic functions perform frequently used mathematical computations. Intrinsic functions should be used in complex calculations rather than the programmer re-writing code that is available for free. This will avoid programming errors, keep the calculation shorter and is self documenting.

6.2.5.3 Nested Computations

DO and IF loops should be indented to improve readability. This is not required by the compiler, but several levels of DO or IF statements not properly aligned is extremely difficult to follow. Additionally, equal levels of complicated calculations should be lined-up vertically with lower case comments, which improves readability and modifiability.

6.2.6 MOE #4 Maintainability

6.2.6.1 Portability

The ability to execute code on several different platforms is desirable. Machine dependent algorithms and routines should be avoided.

6.2.6.2 Memory Management

In FORTRAN, there are many optimization techniques used by the compiler. One way is to reduce I/O system overhead which is controlled by how you set up I/O operations in the source code. For instance, an I/O list consisting of a single unformatted element does not have to be buffered in the Run-Time Library buffers. Also, implied DO loops consisting of a single unnested element are transmitted as a single call to the Run-Time library. To obtain minimum I/O processing, the record length of direct access sequential organization files should be a multiple of the device block size of 512 bytes. Also, memory space can be optimized by controlling data size and code size, dead variable elimination, dead code elimination and elimination of unreachable code.

6.2.6.3 Use of Common Blocks

One of the most effective controls available to the programmer is by the efficient use of COMMON blocks and arrays. The programmer must exercise common sense design techniques such as not defining an overly large array that isn’t used, or declaring variables that are never referenced.

6.2.6.4 Modularity

Each subroutine or function should be short and to the point providing one specific purpose clearly defined and documented.

6.2.6.5 Subroutine Traceability

An important part of documentation is to list in the header other subroutines that are called by the routine you are evaluating, and what their function is. It is also important, when looking at a particular routine, to know what the calling routine was. Being able to understand the program flow from one routine to another, forward and backwards greatly improves modifiability. It is understood that this may be difficult at times since many routines may call the same subroutine, but some type of documentation is desirable.

6.3 SUMMARY OF ANALYSIS AND CONCLUSIONS

6.3.1 Summary

The static analysis conducted here lends itself to compiling metrics on the coding and should provide an indication of how easily others can understand the programmers intent, it provides no information on how well the routines perform. A more dynamic evaluation should be conducted to fully evaluate and verify that the programmers intent is successful upon execution.

Each of the modules within the sample set was critically inspected according to each of the criteria listed beneath each MOE on the Software Evaluation Work Sheet. Observation comments on the compliance levels for each criterion across all the modules in the sample set are summarized in the following sections. The numerical and average scores assigned to the compliance level for each criteria are presented in the next section.

6.3.1.1 MOE #1 Use of Standards

Readability

Although many programmers have contributed to the routines, there is a certain consistency from routine to routine and they were organized in a similar order. Many routines did not contain appropriate white space and section identification that would make the code easier to read. Readability would also be improved if comments were written in a combination of upper and lower case instead of generally all upper case.

Modifiability

The code has a modular structure that generally adheres to good software development practices with meaningful variable names. More descriptive comments are needed to explain specific applications of value-driven and information-oriented processes contained in the code. Modifiability is reduced, for the average programmer, in the flight tactics decision routines due to their specialized, non-technical processes. Also, many routines have multiple entry points and extensive “include” lists that somewhat reduce their modifiability.

ANSI Standards

Each FORTRAN module analyzed was considered to be in compliance with the standard except for the use of the “include” statement, which is non-standard. Many routines contain multiple entry points, which is standard but is not considered a good programming practice. The dummy routines exhibited poor programming practice by not having “return” statements. An example of a statement found in one routine that meets the standard but is considered poor programming practice is “go to (10), icall”. The C modules analyzed were considered to be in compliance with the ANSI C standard.

6.3.1.2 MOE #2 Programming Conventions

Use of Comments and Headers

All modules examined had large descriptive headers containing the version number of the source code complete with authors name and dates that indicated when the code was revised any why. The header also contained a technical description defining the routines purpose and functionality.Sometimes input and output variables, local and external were listed. There was no list of called subroutines. However, some routines did discuss called subroutines in their technical descriptions. Sometimes embedded comments were plentiful and clear while other routines contained too few and marginal comments. The routines did not require classification markings since the program is unclassified.

Use of Formatted Statements

The source code generally contained sections that identified variable declarations, include, TYPE and DATA statements located at the beginning of the module. Some routines mixed the declarations in with the header comments. FORMAT statements were at the end of the module, except in a few isolated cases. Some routines were found to contain FORMAT statements that were not used. Many routines did not have the embedded comments written in lower case with blank lines appropriately placed to highlight certain areas. DO loops and IF blocks were indented to improve readability and understandability.

Logical I/O Devices

All file I/O units were not initialized as variables with no reference to specific units by number. Most of the I/O unit designations used variable names, but several routines contained a mix of variables and specific unit numbers as though they may have been missed during a conversion effort.

Variable Declarations

The variable declaration statements were generally complete and adequate. However, in a few instances defaults were used. None of the modules contained the statement IMPLICIT NONE and a few mixed mode operations were observed.

Variable Initialization

Initializations were in compliance. Some EQUIVALENCE statements were observed, but their use was clear and infrequent. Comments, as always, were sparse.

Variable Naming Conventions

It appeared that variable names followed a naming convention.

Algorithm Clarity

The great majority of the algorithms were developed in a modular fashion and long complicated equations were broken up into smaller sections. A few instances were observed where the algorithms were coded in a confusing manner. No attempt was made to verify the correctness of the algorithms examined.

6.3.2 MOE #3 Computational Efficiency

6.3.2.1 Mixed Mode Calculations

A few mixed mode calculations were observed and evaluated as poor programming practice. There were no comments to describe why the author felt it was necessary to have mixed mode in the computation.

6.3.2.2 Use of Library Functions

The use of library functions was in compliance.

6.3.2.3 Nested Computations

All multiple line DO and IF loops were indented to improve readability.

6.3.3 MOE #4 Maintainability

6.3.3.1 Portability

The machine dependent aspects of "include" statements, specific I/O unit numbers, and routines without "return" statements were not determined by attempting to run BRAWLER on different platforms, but these factors were observed in the static examination of the code. Other than these instances, no other factors were observed that would have an impact on the portability of the code.

6.3.3.2 Memory Management

No first order instances of poor memory management practices were observed. Some routines contain some un-referenced FORMAT statements, which has a very minor impact on memory space usage.

6.3.3.3 Use of Common Blocks

The use of COMMON was in compliance.

6.3.3.4 Modularity

Generally each subroutine or function was short and to the point providing one specific purpose clearly defined and described in the TECHNICAL DESCRIPTION section of the header. However, some routines contained many calls to other routines so that a user has to do a lot of tracing to find all the contributing components comprising some functions.

6.3.3.5 Subroutine Traceability

A desired part of module documentation is a list in the header of other subroutines that are called by the routine and what their function is. None of the headers examined completely listed the routines called. Some contained a list of the major routines called and some included a brief description of their function. Most of the time the called routines discussion was buried in the technical description section of the header and required some digging to locate. A very few of the routines mentioned the calling routine as well.

6.3.4 Scoring Procedure

The code scoring process is a combination of two factors. The first is an evaluation score based on compliance for applicable criterion and the second is that non-applicable criteria are considered to be 100% compliant (i.e., evaluated as excellent). Since most of the routines did not contain all of the evaluation criteria, the excellent score for those not present produces an overall score that tends to indicate a higher level of compliance than what actually exists. The resulting average score for each of the performance levels for each criteria is summarized in Tables 6-2 and 6-3 for all of the non-dummy routines evaluated. Although the dummy routines were not included in the numerical evaluations, their inclusion in the program is considered to be poor programming practice. If there are compelling reasons for their continued inclusion in the program, then the dummy routines should contain explanations in headers that tell what their function should be if they were real routines and why their function has been removed in this version of the program.

TABLE 6-2. BRAWLER V6.2 Evaluation Analysis Average Results.

Criterion	Poor Practice	Acceptable	Excellent
MOE #1 - Use of Standards:			
Criterion #1: Readability		4.50	
Criterion #2: Modifiability		4.51	
Criterion #3: ANSI standards		3.74	
MOE #2 - Programming Conventions:			
Criterion #1: Use of comments and headers		3.35	
Criterion #2: Use of formatted statements		4.91	
Criterion #3: Logical I/O devices		4.71	
Criterion #4: Variable declarations		4.91	
Criterion #5: Variable initialization			5.00
Criterion #6: Variable naming conventions			5.00
Criterion #7: Algorithm clarity		4.91	
MOE # 3 - Computational Efficiency:			
Criterion #1: Mixed mode calculations		4.87	
Criterion #2: Use of library functions			5.00
Criterion #3: Nested computations			5.00
MOE # 4 - Maintainability:			
Criterion #1: Portability			5.00
Criterion #2: Memory management			5.00
Criterion #3: Use of COMMON blocks			5.00
Criterion #4: Modularity			5.00
Criterion #5: Subroutine traceability		3.00	

TABLE 6-3. BRAWLER V6.2 Performance Evaluation Summary.

MOE #1 - Use of Standards	4.25	
MOE #2 - Programming Conventions	4.68	
MOE #3 - Computational Efficiency	4.96	
MOE #4 - Maintainability	4.60	
Average per MOE	4.62	
Overall Evaluation Score (MAX 90)	83.40	92.7% of Max

Table 6-4 shows how the performance level scores were computed. For example, if you take Criterion #1 Readability under MOE #1, 2 of the 111 routines were rated poor practice at 1 point each totaling 2 points, 24 of the 111 routines were rated acceptable at 3 points each totaling 72 points, and 85 of the 111 routines were rated excellent at 5 points each

totaling 425 points. The sum of all the points is 499 divided by 111 routines gives an average performance level rating of 4.50.

TABLE 6-4. BRAWLER V6.2 Evaluation Analysis Scoring Results.

Criterion	Poor Practice	Acceptable	Excellent	Performance Level Score
MOE #1 - Use of Standards:				
Criterion #1: Readability	2*1=2	24*3=72	85*5=425	499/111=4.49
Criterion #2: Modifiability		27*3=81	84*5=420	501/111=4.51
Criterion #3: ANSI standards	1*1=1	70*3=210	41*5=205	416/111=3.74
MOE #2 - Programming Conventions:				
Criterion #1: Use of comments and headers	4*1=4	94*3=282	17*5=85	371/111=3.35
Criterion #2: Use of formatted statements	2*1=2	1*3=3	108*5=540	545/111=4.91
Criterion #3: Logical I/O devices	8*1=8		103*5=515	523/111=4.71
Criterion #4: Variable declarations	1*1=1	3*3=9	107*5=535	545/111=4.91
Criterion #5: Variable initialization			111*5=550	555/111=5.00
Criterion #6: Variable naming conventions			111*5=550	555/111=5.00
Criterion #7: Algorithm clarity	1*1=1	3*3=9	107*5=535	545/111=4.91
MOE # 3 - Computational Efficiency:				
Criterion #1: Mixed mode calculations	3*1=3	1*3=3	107*5=535	541/111=4.87
Criterion #2: Use of library functions			111*5=555	555/111=5.00
Criterion #3: Nested computations			111*5=555	555/111=5.00
MOE # 4 - Maintainability:				
Criterion #1: Portability			111*5=555	555/111=5.00
Criterion #2: Memory management			111*5=555	555/111=5.00
Criterion #3: Use of COMMON blocks			111*5=555	555/111=5.00
Criterion #4: Modularity			111*5=555	555/111=5.00
Criterion #5: Subroutine traceability	7*1=7	97*3=291	7*5=35	333/111=3.00

The sum of the four (4) MOEs totals 83.40 of a possible 90 points. The overall percentage rating is 92.7% (i.e., (83.40/90)*100%). Therefore, for the 111 modules evaluated, on the average they are 92.7% in compliance with the chosen criteria.

6.3.5 Conclusions

The developers of BRAWLER V6.2 adopted many good programming conventions and deserve credit for their efforts in getting a group of programmers over the years to adhere to them. While large descriptive formatted headers were at the beginning of all but 44 of the 111 modules evaluated, they could be improved by containing better variable descriptions, better lists of called subroutines, and for dummy routines an explanation of why the dummy is included. All headers contained a Variable Description heading, but in most cases the section was empty. Some routines used the section to describe only the

parameters named in the call statement. Many GOTO statements were found, some were well documented and clearly understood, but many made the code hard to follow.

Much of the source code was formatted nicely, but additional line spacing would have greatly improved readability. Readability would also be improved if embedded comments were in lower case. The lack of comments on variable names in the header or in the routine body was evident in most of the routines examined. The lack of variable definition makes the code difficult to follow and difficult to modify when required.

Very few of the modules examined listed which subroutines they called, and very few listed the calling routine for the module being examined. It is understood that many routines may call the same module, but to improve modifiability one would like to know the path from one module to another, both forward and back.

The incremental development process applied to BRAWLER over the years is evident in some of the coding in terms of unused format statements, some write statements that use specific unit numbers, and the presence of many dummy routines. It appears that the coding is undergoing upgrading as routines require modification. For the large size and age of the coding in BRAWLER, it is well structured and is remarkably up to date with current practices.

Overall, the basic technical nature of the BRAWLER simulation (an information value-driven, event-decision oriented process) combined with a general lack of variable definitions and poor subroutine traceability make it a difficult program for a new user to understand and to be able to locate specific functions or operations that are being simulated.

While the static analysis conducted here lends itself to determining how the code looks, it provides no information on how well it performs. To fully determine the quality of the software, a dynamic evaluation should be conducted using CASE tools that identify unused and unreachable code as well as providing runtime statistics on call sequences and time spent in routines and can generate various structure charts and tree diagrams.